Software Security 07

Il debugger

Gianluca Amato
Università di Chieti-Pescara

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Il debugging

- Il debugging è il processo che consiste nel *trovare* e *correggere* i bug in un sistema software.
- Il debugging può essere svolto con varie tecniche e strumenti:
  - Test di unità e di integrazione
  - Analisi dei file di output e dei log
  - Profilazione (misura delle risorse utilizzate dal programma)
  - Memory dump
  - Debugger

Il debugger

- Un debugger è uno strumento che consente di
  - Eseguire un programma in condizioni controllate
  - Tracciare le operazioni svolte dal programma
  - Monitorare i cambiamenti delle risorse
  - Visualizzare il contenuto della memoria e dei registri della CPU
  - Modificare il contenuto della memoria e dei registri della CPU

PARTE 1

GDB (GNU Debugger)

GDB: The GNU Debugger

- GDB è un debugger che
  - gira su molti sistemi Unix e similari
  - supporta vari linguaggi di programmazione
    - Assembly (x86, ARM, MIPS)
    - C / C++
    - Rust
    - …
- Normalmente lo usa lo sviluppatore per cercare bug nel programma che lui stesso ha scritto.
- Noi invece lo utilizzeremo per cercare bug in programmi scritti da altri (di cui spesso non abbiamo il sorgente).

Informazioni di debugging

- Per facilitare il debugging, il compilatore può generare delle informazioni addizionali
  - Ad esempio, per ogni funzione, i nomi delle variabili locali e la loro posizione nello stack frame.
- Queste informazioni vengono aggiunte in sezioni speciali del file ELF.
- In `gcc` le informazioni di debugging sono aggiunte usando l'opzione `-g`.

Interfaccia di GDB

- GDB è dotato di una interfaccia testuale a comandi, che può essere lanciata eseguendo il comando `gdb`.

```
$ gdb
GNU gdb (Fedora Linux) 16.2-3.fc42
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Lanciare GDB

- Si può invocare gdb passandogli direttamente il nome del programma del quale effettuare il debugging:
  - `gdb <programma>`
- Si può debuggare un programma che è già in esecuzione:
  - `gdb -p <pid>`

  dove pid (*process id*) è l'identificatore del processo.
- L'opzione `-q` elimina gran parte dei messaggi di avvertimento iniziali.

Comandi

- Un comando di GDB consiste di una singola linea, contenente:
  - il nome del comando;
  - una sequenza di parametri.
- `run`: lancia il programma da debuggare

```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
fib(10) is 55
[Inferior 1 (process 171996) exited normally]
(gdb)
```

Aiuto in linea

- `help`: visualizza una lista di comandi disponibili.

```
(gdb) help
List of classes of commands:

aliases -- User-defined aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
text-user-interface -- TUI is the GDB text based interface.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Argomenti su riga di comando

- `show args`: mostra gli argomenti sulla riga di comando con cui lanciare il programma.
- `set args`: imposta i parametri sulla riga di comando

```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) show args
Argument list to give program being debugged when it is started is "".
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
fib(10) is 55
[Inferior 1 (process 172563) exited normally]
(gdb) set args 20
(gdb) show args
Argument list to give program being debugged when it is started is "20".
(gdb) run
Starting program: /home/amato/fibonacci 20
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
fib(20) is 6765
[Inferior 1 (process 172625) exited normally]
(gdb)
```

Variabili di ambiente

- Questi comandi operano sulle variabili d'ambiente
  - `path <cartella>`
    - aggiunge la cartella al PATH
  - `show paths`
    - mostra il contenuto del PATH
  - `show environment [<var>]:`
    - mostra tutto l'ambiente (o solo la variabile specificata)
  - `set environment <var> <value>`
    - setta il valore per la variable specificata
  - `unset environment <var>`
    - cancella la variabile specificata dall'ambiente

Interrompere l'esecuzione

- La ragione principale per usare un debugger è che possiamo interrompere l'esecuzione di un programma per investigare lo stato della memoria e registri
- In GDB, un programma si può interrompere in 3 situazioni:
  - raggiunge un *breakpoint* (o simile);
  - riceve un *segnale*;
  - termina l'esecuzione passo-passo (*step*) di una istruzione.

Breakpoint e simili

- Un *breakpoint* è un punto del programma impostato in modo che l'esecuzione si interrompa quando viene raggiunto.
  - Un breakpoint può avere ulteriori dettagli che specificano sotto quale condizione il programma si deve interrompere.
- Un *watchpoint* è simile ad un breakpoint
  - interrompe il programma se il valore di una variabile cambia.
- Un *catchpoint* è simile ad un breakpoint
  - Interrompe il programma quando si verificano un certo evento
    - Ad esempio, una chiamata di sistema

Breakpoint in GDB

- In GDB un breakpoint si imposta col comando `break` (o con la sua abbreviazione `b`)
  - `break <location>`
    - imposta un breakpoint alla locazione specificata
  - `break <location> if <cond>`
    - imposta un breakpoint che interrompe il programma solo se la condizione è verificata
- La locazione può essere:
  - Un numero di riga nel programma
  - Il nome di una funzione
  - Un indirizzo in memoria (eventualmente il risultato di una espressione)

Esempio di breakpoint (1)

fibonacci.c

```c
#include <stdio.h>
#include <stdlib.h>

int fibonacci(int n) {
    int k = n;
    if (k <= 2) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main(int argc , char **argv) {
    int n = 10;
    if (argc > 1) {
        n = atoi(argv[1]);
    }
    printf("fib(%d) is %d\n", n, fibonacci(n));
}
```

Esempio di breakpoint (2)


```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

Esempio di breakpoint (2)

compila con dati di debug

```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

Esempio di breakpoint (2)

esegue gdb

```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

Esempio di breakpoint (2)

imposta argomenti

```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

Esempio di breakpoint (2)


```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

aggiunge un breakpoint condizionale

Esempio di breakpoint (2)


```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

esegue il programma

Esempio di breakpoint (2)


```
amato@banzai:~$ gcc -g -o fibonacci fibonacci.c
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) set args 5
(gdb) break fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb)
```

il programma si interrompe quando viene raggiunto il breakpoint e la condizione è vera

Continuazione ed esecuzione passo-passo

- Quando un programma si interrompe, la sua esecuzione può essere ripresa in due modi:
  - *Continuando* l'esecuzione (comando `continue` / `c`)
    - il programma riprende l'esecuzione finché non viene interrotto nuovamente o termina regolarmente.
  - Eseguendo *una singola istruzione* del programma
    - comando `step` / `s`
      - se l'istruzione è una chiamata di funzione, si ferma appena all'inizio della funzione
    - comando `next` / `n`
      - Se l'istruzione è una chiamata di funzione, la esegue senza interruzione e si ferma solo quando ritorna

Esempio comandi di continuazione


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b main
Breakpoint 1 at 0x4004c8: file fibonacci.c, line 14.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdac8) at fibonacci.c:14
14          int n = 10;
(gdb) step
15          if (argc > 1) {
(gdb) step
18          printf("fib(%d) is %d\n", n, fibonacci(n));
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 228102) exited normally]
(gdb)
```

viene eseguita una singola istruzione

Esempio comandi di continuazione


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b main
Breakpoint 1 at 0x4004c8: file fibonacci.c, line 14.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdac8) at fibonacci.c:14
14          int n = 10;
(gdb) step
15          if (argc > 1) {
(gdb) step
18          printf("fib(%d) is %d\n", n, fibonacci(n));
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 228102) exited normally]
(gdb)
```

ancora una singola istruzione

Esempio comandi di continuazione


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b main
Breakpoint 1 at 0x4004c8: file fibonacci.c, line 14.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdac8) at fibonacci.c:14
14          int n = 10;
(gdb) step
15          if (argc > 1) {
(gdb) step
18          printf("fib(%d) is %d\n", n, fibonacci(n));
(gdb) continue
Continuing.
fib(10) is 55
[Inferior 1 (process 228102) exited normally]
(gdb)
```

viene eseguita una singola istruzione

continuiamo fino alla fine del programma o ad un breakpoint

Ispezionare lo stack

- Mentre il programma è interrotto la prima cosa che ci interessa sapere è perché si è interrotto e come siamo arrivati lì.
  - queste informazioni si trova nella pila dei record di attivazione.
- I seguenti comandi si possono usare per leggere lo stack:
  - `frame <selezione>`
    - Stampa una breve descrizione dello stack frame selezionato;
  - `info frame <selezione>`
    - Stampa una descrizione dettagliata dello stack frame selezionato.

Esempio di ispezione dello stack


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame
#0  fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame 1
#1  0x00000000004004a2 in fibonacci (n=4) at fibonacci.c:9
9               return fibonacci(n - 1) + fibonacci(n - 2);
```

il comando `frame` ci da una breve descrizione del frame corrente

Esempio di ispezione dello stack


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame
#0  fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame 1
#1  0x00000000004004a2 in fibonacci (n=4) at fibonacci.c:9
9               return fibonacci(n - 1) + fibonacci(n - 2);
```

numero del frame corrente

Esempio di ispezione dello stack


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame
#0  fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame 1
#1  0x00000000004004a2 in fibonacci (n=4) at fibonacci.c:9
9               return fibonacci(n - 1) + fibonacci(n - 2);
```

nome della funzione, argomenti e linea di codice

Esempio di ispezione dello stack


```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) b fibonacci if n==3
Breakpoint 1 at 0x400482: file fibonacci.c, line 5.
(gdb) run
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame
#0  fibonacci (n=3) at fibonacci.c:5
5           int k = n;
(gdb) frame 1
#1  0x00000000004004a2 in fibonacci (n=4) at fibonacci.c:9
9               return fibonacci(n - 1) + fibonacci(n - 2);
```

Informazione per il frame precedente (sempre `fibonacci` ma con n=4)

Esempio di ispezione dello stack

il comando `info frame` ci da una descrizione dettagliata del frame corrente

```
(gdb) info frame
Stack level 1, frame at 0x7fffffffd800:
 rip = 0x4004a2 in fibonacci (fibonacci.c:9); saved rip = 0x4004a2
 called by frame at 0x7fffffffd840, caller of frame at 0x7fffffffd7c0
 source language c.
 Arglist at 0x7fffffffd7f0, args: n=4
 Locals at 0x7fffffffd7f0, Previous frame's sp is 0x7fffffffd800
 Saved registers:
  rbx at 0x7fffffffd7e8, rbp at 0x7fffffffd7f0, rip at 0x7fffffffd7f8
```

Il comando `start`

- Siccome la combinazione
  - `break main`
  - `run`

  è piuttosto comune, se si vuole eseguire un programma dall'inizio si può usare il comando `start`.

Disassembly

- Il comando `disassemble` può essere usato per disassemblare una funzione.
- Si noti che utilizza la notazione AT&T. Se si vuole usare la notazione Intel:
  - `set disassembly-flavor intel`

```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) start
Temporary breakpoint 1 at 0x4004c8: file fibonacci.c, line 14.
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdac8) at f
14          int n = 10;
(gdb) disas
Dump of assembler code for function main:
   0x00000000004004b9 <+0>:     push   %rbp
   0x00000000004004ba <+1>:     mov    %rsp,%rbp
   0x00000000004004bd <+4>:     sub    $0x20,%rsp
   0x00000000004004c1 <+8>:     mov    %edi,-0x14(%rbp)
   0x00000000004004c4 <+11>:    mov    %rsi,-0x20(%rbp)
=> 0x00000000004004c8 <+15>:    movl   $0xa,-0x4(%rbp)
   0x00000000004004cf <+22>:    cmpl   $0x1,-0x14(%rbp)
   0x00000000004004d3 <+26>:    jle    0x4004eb <main+50>
   0x00000000004004d5 <+28>:    mov    -0x20(%rbp),%rax
   0x00000000004004d9 <+32>:    add    $0x8,%rax
   0x00000000004004dd <+36>:    mov    (%rax),%rax
   0x00000000004004e0 <+39>:    mov    %rax,%rdi
   0x00000000004004e3 <+42>:    call   0x400380 <atoi@plt>
   0x00000000004004e8 <+47>:    mov    %eax,-0x4(%rbp)
   0x00000000004004eb <+50>:    mov    -0x4(%rbp),%eax
```

Esaminare i dati

- La maniera standard per esaminare i dati nel vostro programma è usare il comando
  - `print` / `p`

  o il suo sinonimo
  - `inspect`.
- Si può ispezionare il valore dei registri con
  - `info registers`.

```
amato@banzai:~$ gdb -q fibonacci
Reading symbols from fibonacci...
(gdb) start
Temporary breakpoint 1 at 0x4004c8: file fibonacci.c, line 14.
Starting program: /home/amato/fibonacci
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdac8) at fi
14          int n = 10;
(gdb) print n
$1 = 32767
(gdb) step
15          if (argc > 1) {
(gdb) print n
$2 = 10
(gdb) info registers
rax            0x7ffff7fa0e28      140737353748008
rbx            0x0                 0
rcx            0x402e00            4206080
rdx            0x7fffffffdad8      140737488345816
rsi            0x7fffffffdac8      140737488345800
rdi            0x1                 1
rbp            0x7fffffffd9a0      0x7fffffffd9a0
rsp            0x7fffffffd980      0x7fffffffd980
r8             0x7ffff7f99680      140737353717376
r9             0x7ffff7f9b000      140737353723904
r10            0x7fffffffd6e0      140737488344800
```

GDB per programmi senza sorgente

- GDB è pensato soprattutto per il debugging di:
  - codice sorgente…
  - …compilato con simboli di debugging
- Lo si può usare per *reverse engineering* e *sviluppo di exploit*, ma l'interfaccia utente lo rende scomodo:
  - ad esempio, bisogna continuamente usare comandi come `disas` e `info registers` per esaminare il programma in linguaggio macchina in esecuzione e il valore dei registri.

Comandi `stepi` e `nexti`

- I comandi `step` e `next` non funzionano in maniera corretta in mancanza di sorgente. Ma ci sono delle alternative:
  - `stepi` / `si` : esegue una singola istruzione in LM
  - `nexti` / `ni` : esegue una singola istruzione in LM, ma non entrare dentro il codice di una funzione (istruzione `call`)

Demo

Svolgere le challenge

Software 12 – Dynamic 5

Esercizi

Svolgere le challenge

Software 13 – Dynamic 6
Software 14 – Dynamic 7
Software 15 – Dynamic 8
Software 16 – Dynamic 9

(si tratta di challenge estremamente guidate, come la Software 12)

GEF: GDB Enhanced Features

GEF: GDB Enhanced Features

- GEF è un plugin per GDB che aggiunge:
  - Una *dashboard* che mostra continuamente lo stato di registri e memoria.
  - La *dereferenziazione* automatica di puntatori, siano essi in dati o registri.
  - Analisi dell'heap.
  - Informazioni sul file ELF.
- È basata sulla API Python di GDB.

GEF: GDB Enhanced Features (2)

- GEF è stato sviluppato con i seguenti obiettivi
  - Semplice e veloce da installare
  - Nessuna dipendenza esterna
  - Indipendente dall'architettura software
  - Estendibile
  - Ben documentato

Installazione di GDB-GEF

- Per i sistemi Fedora e derivati:
  - `dnf install gdb-gef`
    - Il comando `gdb` lancia il debugger standard
    - Il comando `gdb-gef` lancia GDB con l'esternsione GEF
- Per i sistemi Debian e derivati
  - Non esiste un pacchetto già pronto, usare le istruzioni che trova nel sito git di gdb-gef o nella documentazione.

Alcuni comandi di GEF (1)

- Stampa di registri con *dereferenziazione* automatica

```
gef➤  registers
$zero: 0x0
$at  : 0x1
$v0  : 0x77ff9490
$v1  : 0x7fffe6c8  →  0x77e61498  →  <__libc_start_main+200> bnez v0, 0x77e614f0 <__libc_start_main+288>
$a0  : 0x1
$a1  : 0x7fffe784  →  0x7fffe880  →  "/home/user/simple-bof"
$a2  : 0x7fffe78c  →  0x7fffe896  →  "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so[...]"
$a3  : 0x0
$t0  : 0x77e613d0  →  <__libc_start_main+0> lui gp, 0x17
$t1  : 0x80808080
$t2  : 0xb64
$t3  : 0x0
$t4  : 0x3
$t5  : 0x0
$t6  : 0x77fff7f0  →  0x77fcc000  →  0x464c457f
$t7  : 0x55555704  →  <hlt+0> b 0x55555704 <hlt>
$s0  : 0x0
$s1  : 0x555559f0  →  <__libc_csu_init+0> lui gp, 0x2
$s2  : 0x77ffeedc  →  0xbafeb100
$s3  : 0x0
```

Alcuni comandi di GEF (2)

- Informazione sullo *heap*

```
gef➤  heap chunks
Chunk(addr=0x555555756010, size=0x250, flags=PREV_INUSE)
    [0x0000555555756010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x555555756260, size=0x110, flags=PREV_INUSE)
    [0x0000555555756260     61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa]
Chunk(addr=0x555555756370, size=0x110, flags=PREV_INUSE)
    [0x0000555555756370     62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62    bbbbbbbbbbbbbbbb]
Chunk(addr=0x555555756480, size=0x110, flags=PREV_INUSE)
    [0x0000555555756480     63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63    cccccccccccccccc]
Chunk(addr=0x555555756590, size=0x110, flags=PREV_INUSE)
    [0x0000555555756590     64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64    dddddddddddddddd]
Chunk(addr=0x5555557566a0, size=0x20970, flags=PREV_INUSE)  ←  top chunk
```

Alcuni comandi di GEF (3)

- Informazioni sulla sicurezza.

```
gef➤  checksec
[+] checksec for '/home/user/mips-stack-bof'
Canary                        : No
NX                            : No
PIE                           : Yes
Fortify                       : No
RelRO                         : No
gef➤
```

Alcuni comandi di GEF (4)

- Dump esadecimale della memoria

```
gef➤  db $sp
0x00007fffffffdfa0     60 62 75 55 55 55 00 00 70 63 75 55 55 55 00 00     `buUUU..pcuUUU..
0x00007fffffffdfb0     80 64 75 55 55 55 00 00 90 65 75 55 55 55 00 00     .duUUU...euUUU..
0x00007fffffffdfc0     30 47 55 55 55 55 00 00 97 5b a0 f7 ff 7f 00 00     0GUUUU...[......
0x00007fffffffdfd0     01 00 00 00 00 00 00 00 a8 e0 ff ff ff 7f 00 00     ................
```

Estensioni di GEF

- Il pacchetto GEF-Extras contiene estensioni di GEF, tra cui:
  - **assemble**
    - Assemblatore basato su Keystone
  - **capstone**
    - Disassemblatore alternativo a quello integrato basato su capstone
  - **emulate**
    - Esecuzione delle istruzioni in ambiente emulato basato su unicorn
  - **ropper**
    - Generatore di gadget per ROP (Return Oriented Programming)
  - **retdec**
    - Decompilatore basato su retdec

Demo

Svolgere le challenge

SS_0.04 – volatility

SS_1.01 – NextGen Safe

Tecniche di anti-debugging

Tecniche di anti-debugging

- Un programma può cercare di impedire l'analisi dinamica tramite tecniche di anti-debugging
  - L'idea è di scrivere il programma in maniera tale che si generi un errore se si cerca di tracciare l'esecuzione del programma.
  - Uno degli approcci più semplici è quello noto come *self-debugging*.
- Esempio di challenge che usa il self-debugging:

Self debugging

- Viene creato un nuovo processo per tracciare l'esecuzione del nostro programma.
  - Si utilizza a tal scopo la funzione `ptrace`.
- Se c'è già un debugger in esecuzione, la funzione genera un errore e il programma termina.

Esempio di self-debugging


```
int main(int argc, char** argv) {
        if (ptrace(PTRACE_TRACEME,0,1,0) == -1) {
                return -1;
        }
        char* name = "Calef";
        char* input = malloc(10);

        printf("Guess my name: ");
        scanf("%10s",input);
        if (strcmp(name,input)==0) {
                printf("Well done! You have guessed my name!\n");
        } else {
                printf("I am sorry, this is not my name!\n");
                printf("Please, try again.\n");
        }
}
```

Altre tecniche di anti-debugging

- Esistono altre tecniche di anti-debugging.
  - Ad esempio, è possibile misurare il tempo che viene impiegato nell'esecuzione di una funzione.
  - Se il tempo è eccessivo, è perché qualcuno sta eseguendo il programma passo passo.
- La challenge SS_1.06 – pacman utilizza sia il self-debugging che la tecnica basata sul tempo di esecuzione.

Gianluca Amato
Università di Chieti-Pescara

Software Security 07

Il debugger

FINE